Automatic generation of software-based functional failing test for speed debug and on-silicon timing verification

E. Sanchez [*], G. Squillero [*], A. Tonda [§]

[*] Politecnico di Torino, Torino, Italy — { ernesto.sanchez, giovanni.squillero }@polito.it
[§] Institut des Systèmes Complexes, Paris, France — alberto.tonda@iscpif.fr

Abstract— **The 40 years since the appearance of the Intel 4004 deeply changed how microprocessors are designed. Today, essential steps in the validation process are performed relying on physical dices, analyzing the actual behavior under appropriate stimuli. This paper presents a methodology that can be used to devise assembly programs suitable for a range of *on-silicon* activities, like speed debug, timing verification or speed binning. The methodology is fully automatic. It exploits the feedback from the microprocessor under examination and does not rely on information about its microarchitecture, nor does it require design-for-debug features. The experimental evaluation performed on a Intel Pentium Core i7-950 demonstrates the feasibility of the approach.**

Index Terms— **Microprocessor, Speed debug, On-silicon verification, Software-based functional failing test, Evolutionary algorithm.**

I. Introduction

Post-silicon validation is the tip of the iceberg of a growing trend. Nowadays, significant steps in the validation of microprocessors must be performed *on silicon*, i.e., running experiments on physical devices after tape-out. The cost of manufacturing prototypes is enormous, but often there are no alternatives to meet the market demand.

For example, the typical design flow of a modern microprocessor goes through several iterations of frequency pushes prior to final volume production. The process is sometimes referred to as *speed stepping*: a prototype is tested at increasing clock frequencies until a misbehavior is detected. The problem is then analyzed, and eventually traced back to the design. As soon as a speed grade is reached with good timing yield, the target is set for an even higher speed grade [1].

The identification of paths that actually limit the performance of a chip is called *speed debug*. It is of paramount importance because such paths may be the locations where design fixes should be applied, or they may indicate holes in the design methodologies. Indeed, speed debug, as well as other timing verification tasks, must be performed on silicon. In nanometer processes it is not feasible to consider simultaneously all factors that contribute to the timing behavior during the pre-silicon analysis, and even the analysis algorithms themselves are often approximated or oversimplified [2] [3].

A *failing test* is a pattern of operations that uncovers an incorrect behavior. The availability of failing tests is essential for performing an effective speed debug. Unfortunately, the development of failing tests can be very expensive and time consuming. A *software-based functional failing test* is an assembly-language program whose result is functionally incorrect. That is, the misbehavior may be detected simply checking the values in the registers at the end of the execution. Differently from *functional tests*, as defined in [4], a software-based functional failing test does not require an expensive test tester to be applied, nor any *design-for-debug* circuit features.

This paper proposes a methodology for the automatic generation of software-based functional failing tests suitable for speed debug, speed binning or other on-silicon activities. The methodology exploits the functional results calculated by running candidate tests to generate more and more efficient tests. The proposed approach does not rely on the knowledge of the microarchitecture of the device under test, nor on presumptive information about its internal design.

A feasibility study of the methodology has been recently presented in a poster at VLSI-SoC [5]. The study exploited *undervolting* to simulate the instabilities caused by an increase in operating frequency, and a simpler system to run experiments. This paper describes the real system, tackling directly the operating frequency. The results gathered on an Intel Pentium Core i7-950 clearly demonstrate the feasibility and effectiveness of the approach.

Section II outlines the proposed approach and section III details how code is optimized. Section IV explains the experimental evaluation. Section V concludes the paper.

II. Proposed Methodology

The proposed approach can be classified as *feedback-based*: candidate failing tests are created without a rigid scheme, and evaluated on the target microprocessor. The data gathered are fed back to the generator and used to generate a new, enhanced set of candidate tests. The process is then iterated while improvements are achieved.

Two computers are used: the *master* computer runs the optimizer and creates the candidate tests; the *slave* executes them. The master also controls the operating frequency of the slave (see Figure 1).

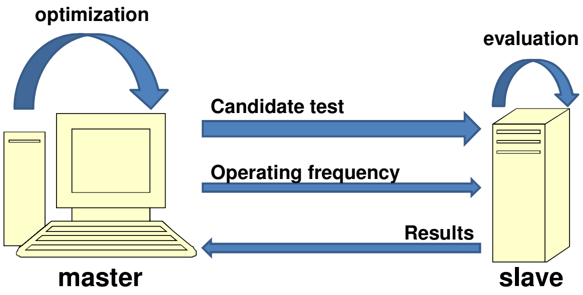

Figure 1: System architecture

A functional failing test for speed debug is an assembly-language program that produces the correct result only while the microprocessor operating frequency is below a certain threshold. Let us denote this threshold as its *functional frequency threshold*, because the incorrect behavior is functionally observable.

978-0-7695-4594-3/11 $26.00 © 2011 IEEE
DOI 10.1109/MTV.2011.19

Once the slave executes the program, the most relevant feedback extracted is the functional frequency threshold of the candidate test. That is, the frequency when the result ceases to be correct.

The practical usefulness of a functional speed-path failing test increases as its functional frequency threshold decreases: a test that produces a failure at a relatively low frequency is preferable to a test that fails only at very high frequencies.

Non-deterministic effects pose additional challenges: design criticalities may appear only occasionally and possibly only in a percentage of the manufactured chips. Thus, the test is repeated several times on the slave, and the results sent back to the master. The practical usefulness of a functional speed-path failing test increases with its predictability: a test failing half of the times is more useful than a test that produces a single failure every thousand runs.

The last component of the feedback is the actual number of incorrect behaviors.

III. Failing Test Optimizer

The core of the optimizer inside the master is an evolutionary algorithm, that is, a software that loosely mimics some principles of the Neo-Darwinian paradigm, namely variation, inheritance, and selection.

The toolkit exploited in this work is called µGP (also known as *MicroGP*) [6], it is freely available under the *GNU Public License* from *Sourceforge*[1]. Since it has already been used in several works, its description is out of the scope of this text. However, it could be useful to remind that µGP is an evolutionary optimizer. Natural evolution is not a random process. On the contrary, it is based on random variations, but some are rejected while others preserved according to objective evaluations. Only changes that are beneficial to the individuals are likely to spread into subsequent generations. Darwin called this principle "natural selection" [7]. When natural selection causes variations to be accumulated in one specific direction the result strikingly resembles an optimization process, and µGP takes advantage of it. In applications like the one presented here, it is able to optimize solutions only requiring to assess the *effect* of random changes, not the ability to *design* intelligent modifications.

In µGP candidate test programs are encoded as directed multigraphs. During the optimization process, a test program undergoes several types of modifications that ape both sexual and asexual reproduction. For example one or more instructions can be added; one or more instructions may be removed; the operands of certain instructions can be modified. New programs may also be obtained by mating existing ones, and the multigraph representation ensures that the offspring is still a sensible program, resembling both parents and, thus, inheriting potentially good characteristics from both of them.

Modifications are completely random, with the only judiciousness of being small changes more probable than large ones. However, the evaluation of the candidate solutions is objective, and, generation after generation, good characteristics are preserved, while useless one are discarded. As a result, candidate solutions are optimized "through the accumulation of slight but useful variations", in Darwin own words. Indeed, the ability of similar tools to stress microarchitectural features of a microprocessor was already demonstrated in the past, tackling the post-silicon validation of an Intel Pentium 4 [8].

The efficacy of the evolutionary core depends on several factors. The most important ones are: how candidate solutions are evaluated; and what is encoded inside individuals. Exploiting an evolutionary approach is per-se of little interest, while tuning such elements can be the key factor for effectively finding a solution.

A. Fitness Function

Evolutionary algorithm scholars call *fitness* the goodness of a solution. Artificial evolution, as well as the natural one, is based on the idea of *differential survival*. That is, different individuals must have a different chance to survive. And, in order to be distinguishable in the artificial environment, they must have a different fitness.

As said before, the first and most important component of the feedback is the functional frequency threshold. The second is the number of failures detected over the R repetitions at the functional frequency threshold. The last is the number of incorrect results recorded during the running of the test.

Similarly to *software-based self test* [9], candidate test programs include a mechanism for checking their correctness: all the results of the calculations performed by the test program are compacted in a single signature using a hash function. Details about the hash itself will be discussed later.

The master computer first runs the test program and stores the signature. Then it runs the program on the slave computer at increasing operating frequencies, checking that the signature is not modified. As soon as a difference is detected, the functional frequency threshold is recorded. The whole process is repeated R times to tackle variability.

Operatively, µGP creates assembly functions that are assembled and linked with a *manager* module. These functions contain a loop that executes L times a set of instructions. The instructions themselves are devised by the evolutionary core, while the framework is fixed. At the end of the loop, before the next iteration, the values in the registers are used to update the signature.

B. Internal Representation, Multithreading and Multicore

The internal representation is another key aspect. The evolutionary algorithm must be given the opportunity to generate useful solutions. Modern processors may implement a multithreaded design; or they can exploit a multicore architecture; or even both. A single individual is composed of different independent functions.

Inside each thread, the assembly instructions made available to µGP can be divided in three main classes: *integer instructions*; legacy *x87 instructions*; *single-instruction/multiple-data* (SIMD) *instructions*. Not surprisingly, SIMD instructions are particularly critical during speed stepping: the complex calculations involved by these instructions cause data to go through several functional units, and the resulting *datapaths* are prone to be source of problems when the operating frequency is increased.

Cache memories must be taken into account as well, since there may be a significant difference in performance and power consumption between a L1 cache hit and a L1 cache miss. µGP was given the possibility to generate cache hits and cache misses through a set of variables carefully spaced in memory.

It must be noted that the goal of adding such variables is to let the evolutionary core control the cache activity, but no suggestions are given on how to exploit them. µGP finds autonomously which sequence of operations is more useful to generate a failing test.

[1] http://ugp3.sourceforge.net/

IV. Experimental Evaluation

Devising a comparison for the proposed methodology is not an easy task: there are no publicly-available test suites for assessing results on functional failing-test generation, and very few results have been disclosed in the scientific literature. Neither is possible to compare against works in delay testing for speed binning, such as [4], because no structural information about the tested units is available to the public.

However, a closely related problem is frequently faced by the *overclockers*, a community of computer enthusiasts. Overclockers enjoy themselves pushing the performance of their microprocessors by increasing the operating frequency far beyond the nominal specification [10]. For instance, an Intel Celeron D 352 has been reported running with a clock above 8.3 GHz[2], more than twice the nominal 3.2 GHz.

After reaching such frequencies, overclockers need to assess the stability of their systems. The whole community is actively seeking *stability tests* able to quickly and reliably discriminate a working system from one that have been pushed too far. Such test suites are used to stress the systems and highlight criticalities, thus they may be regarded as generic functional fail tests not focused on a specific microprocessor. They have been used as a baseline to evaluate the performances of the proposed methodology.

While all the stability tests are quite different, a common point is that modern ones do extensive SIMD calculation. Another common point is their ability to increase the temperature of the microprocessor. It is well known that high temperature may cause both reversible and irreversible effects on electronic devices. Heating may increase the skew of the clock net and alter hold/setup constraints, causing design criticalities to become manifest and the circuit to operate incorrectly [11].

However, while such an effect is sensible when assessing the stability of a system, it may not be desirable when the goal is to find a failing test during speed stepping. The main reason is that the failing test should be as repeatable as possible, while increasing the temperature also increases non-deterministic phenomena. Nevertheless, since no other comparison is possible, the proposed approach was tested against the state-of-the-art stress tests used by the overclocking community.

A. Overclockers' Stress Tests

Most of the information about stability stress tests is available through forums and web sites on the internet, with few or none official sources. However, there is quite a generalized agreement in the overclockers community on these tools.

Prime95 is the name of an application written by George Woltman and used by a project for finding *Mersenne prime numbers*[3]. It makes extensive use of the Fast Fourier Transform, or FFT, with a highly efficient implementation that exploits SIMD instructions. Over the years, it has become extremely popular among overclockers as a stability test. It includes a “Torture Test” mode designed specifically to test systems and highlight problems. In the overclocking community, the rule of thumb is to run it for some tens of hours.

LINPACK is a software library for performing numerical linear algebra on digital computers. It was originally written in Fortran in the 1970s and early 1980s. Newer implementations of LINPACK exploit SIMD instructions and are highly optimized. Significantly, Intel includes a benchmark based on an optimized version of LINPACK in its *Math Kernel Library*[4]. Different applications exploited such benchmark to assess the stability. The most common are *LinX*[5], *IntelBurnTest*[6], and *OCCT*[7]. The last one also includes a proprietary stress test.

B. Target System

Experiments were run on an Intel Pentium Core i7-950 on an *ASUS* motherboard *Rampage III Extreme* with the Intel chipset *X58*. The system was equipped with 6 GiB RAM memory DDR3 1600 MHz, and a *Radeon HD 5870* graphic card. The default clock ranges between 3.06GHz and 3.48GHz, thanks to the so-called *Intel turbo boost technology 2.0* that automatically allows processor cores to run faster than their base operating frequency.

TABLE I. µGP PARAMETERS

Parameter	Meaning	Value
µ	Size of the population	30
ν	Size of the initial (random) population	100
λ	Operators applied in each generation	20
R	Repetitions of each test to tackle variability	10
L	Repetitions inside each test	5,000,000
C	Variables to exploit cache hit/miss	16

The *i7-950* is based on the *Nehalem* architecture, the successor of the *Core* architecture. It supports the *SSE 4.2* instructions, adding 7 new instructions to the *SSE 4.1* set available in the *Core 2* series. It is a quad-core microprocessor, able to run up to 8 threads with simultaneous multithreading. Each core has two separate 32 KiB L1 caches for data and instructions, both implementing an 8-way set associative architecture. Each core has also an L2 cache of 1 MiB, 8-way set associative that is used for both data and instructions. There is an additional 8 MiB L3 cache, 16-way set associative that is shared by the 4 cores using a design branded as *Intel smart cache*.

C. Experimental Results

The failing tests devised by the proposed approach for the target microprocessor running at frequencies higher than the nominal one were compared with the state-of-the-art stress tools used by the overclocking community. Results obtained with different V-cores are reported in Table II and Table III. µGP required about 100 hours to generate each failing test. Adopted parameters are shown in Table I. The first three (µ, ν and λ) control the evolutionary engine and are de-facto standard. *R* and *L* control the evaluation of candidate tests. The last one (*C*) limits the possibility to create cache hit/miss.

Columns are labeled with the name of the program used to test the system. The last column reports data of the test generated by µGP. Rows indicate the CPU core voltage at which the experiments were run. Cells shows the time required for the given stress test to report a failure either in seconds (“) or minutes (‘). To reduce overheating effects, all tests were stopped after 10 minutes. The infinity sign “∞”

[2] See http://valid.canardpc.com/records.php

[3] A *Mersenne number* is a positive integer that is one less than a power of two: $M = 2^p - 1$. Remarkably, the largest known prime number is a Mersenne prime number: $N = 2^{43,112,609} - 1$.

[4] http://software.intel.com/en-us/intel-mkl/

[5] Originally posted on http://forums.overclockers.ru/

[6] http://www.ultimate-filez.com/

[7] http://www.ocbase.com/perestroika_en/

means that no failure has been detected in the allowed time. It must be noted that the µGP-designed test program does the first check of the signature about 30 seconds after it starts. All experiments have been repeated 10 times.

TABLE II. TIME REQUIRED TO DETECT AN INCORRECT BEHAVIOR (V-CORE SET TO 1.24375 VOLT).

CPU Freq. [GHz]	IBT	LinX	OCCT	Prime95	µGP
3.827	2’	3’	5’	30”	29”
3.803	10’	4’	∞	9’	29”
3.783	∞	5’	∞	∞	29”
3.758	∞	6’	∞	∞	29”
3.737	∞	∞	∞	∞	30”
3.721	∞	∞	∞	∞	30”
3.691	∞	∞	∞	∞	30”
3.666	∞	∞	∞	∞	30”
3.645	∞	∞	∞	∞	77”
3.622	∞	∞	∞	∞	∞

Failing tests devised with the proposed methodology clearly outperform all the other approaches, forcing the processor to fail at frequencies which are significant lower than all other. Remarkably, µGP was asked to find a very fast failing test for a specific microprocessor, and therefore there is no guarantee that the devised program would fail on a different model. Moreover, the test was required to be very short, to avoid heating effects. On the contrary, stress tests intentionally exploit overheating and are designed to work with different architectures.

TABLE III. TIME REQUIRED TO DETECT AN INCORRECT BEHAVIOR (V-CORE SET TO 1.2500 VOLT)

CPU Freq. [GHz]	IBT	LinX	OCCT	Prime95	µGP
3.827	6’	6’	8’	3’	28”
3.803	∞	6’	∞	∞	28”
3.783	∞	∞	∞	∞	29”
3.758	∞	∞	∞	∞	29”
3.737	∞	∞	∞	∞	29”
3.721	∞	∞	∞	∞	30”
3.691	∞	∞	∞	∞	30”
3.666	∞	∞	∞	∞	30”
3.645	∞	∞	∞	∞	∞
3.622	∞	∞	∞	∞	∞

It must also be noted that the temperature of the microprocessor during the experiments never exceeded 50°C, while it was significantly higher while running LINPACK-based stress tests, even with the liquid cooling.

Figure 2 shows the code generated for one thread of the i7-950. The fragment is inserted in a fixed schema and executed a given number of times. The values of all registers is saved into an hash at the end of each loop. All the variables labeled with *v* are likely to be cached on the same L1 line. SIMD instructions have been clearly favored by the optimization process, however, not knowing the underlying microarchitecte, further examinations of the code is beyond our possibilities.

D. Feedback from the overclockers community

The generated tests were made available to the overclockers community as *ultra-fast stability test*[8]. The feedback is summarized in Table IV. The column CPU shows the CPU model used in the experiments. The two columns labeled with *Frequency* report the nominal (N) frequency of the CPU and the one actually used by the overclocker (A). The next column shows the actual V-Core. The following columns report the results of the various stability test: IBT, LinX, OCCT and the one generated by µGP. All the programs were considered *stability tests*, thus “FAIL” is a positive result, meaning that the test was able to uncover the instability. On the other hand, “PASS” means that the test was unable to pinpoint any problem.

Some overclockers did not run comparison tests with IBT, LinX or OCCT. Nevertheless, the fact that they try the µGP one implies that they were considering their system fully reliable.

TABLE IV. FEEDBACK FROM THE OVERCLOCKERS COMMUNITY

CPU	Frequency		V-Core	IBT	LinX	OCCT	µGP
	N	A					
i7 860	2.80	4.25	1.4	-	**FAIL**	-	PASS
i7 860	2.80	4.30	1.4	-	**FAIL**	-	**FAIL**
i7 920	2.66	4.02	1.27	-	PASS	-	**FAIL**
i7 920	2.67	2.65	1.27	-	-	-	PASS
i7 920	2.67	3.20	1.0	PASS	-	-	**FAIL**
i7 920	2.67	3.20	1.044	PASS	-	-	PASS
i7 920	2.67	3.20	1.0312	**FAIL**	-	-	**FAIL**
i7 920	2.67	3.20	1.0375	**FAIL**	-	-	**FAIL**
i7 920	2.67	4.20	1.35	-	-	-	PASS
i7 920	2.67	4.33	1.385	-	PASS	-	**FAIL**
i7 920	2.67	4.40	1.45	-	-	-	PASS
i7 930	2.80	3.80	1.2	-	-	-	PASS
i7 950	3.06	4.03	1.31	PASS	-	-	PASS
i7 950	3.06	4.03	1.28	**FAIL**	-	-	**FAIL**
i7 950	3.06	4.03	1.328	PASS	-	PASS	**FAIL**
i7 950	3.06	4.20	1.34	PASS	PASS	-	**FAIL**
i7 950	3.06	4.20	1.31	PASS	PASS	-	**FAIL**
i7 965	3.20	3.46	1.21	-	-	-	PASS

Although not systematic, the feedback fully confirmed our claims: results on i7-950 microprocessors show the superiority of the µGP test. Similar results are achieved on all i7-9xx units. Interestingly, the failing test is not effective on the i7-860 family. Thus, it sounds plausible that the test stresses specific microarchitectural features present in the former families but not in the i7-8xx one.

V. CONCLUSIONS

The paper proposed an efficient post-silicon methodology for devising software-based functional failing tests. Such failing test may be exploited during speed debug or other on-silicon activities, like timing verification.

Experimental results clearly demonstrate that tests are able to highlight criticalities very specific of the target microarchitecture. More interestingly, it is able to do it without any information about the design. The methodology was successfully tested on an Intel Pentium Core i7-950 and could be very easily applied to different devices.

The proposed methodology could be easily exploited by microprocessor manufacturers during timing verification, speed debug or other post-silicon activities.

VI. ACKNOWLEDGEMENTS

The authors would like to thank Maria Di Vincenzo, Gianluca Fisanotti and Giuseppe Parlato for implementing the approach, brilliantly solving several practical problems, and running the experiments.

[8] http://www.cad.polito.it/research/Evolutionary_Computation/Overclocking.html

VII. REFERENCES

[1] J. Zeng, R. Guo, W.-T. Cheng, M. Mateja, and J. Wang, "Scan-based Speed-path Debug for a Microprocessor," *IEEE Design and Test of Computers*, Jun. 2011.

[2] K. Killpack, C. Kashyap, and E. Chiprout, "Silicon Speedpath Measurement and Feedback into EDA flows," in *44th Design Automation Conference*, 2007, pp. 390-395.

[3] N. Callegari, L. .-C. Wang, and P. Bastani, "Speedpath analysis based on hypothesis pruning and ranking," in *46th ACM/IEEE Design Automation Conference*, 2009, pp. 346-351.

[4] J. Zeng, et al., "On correlating structural tests with functional tests for speed binning of high performance design," in *International Test Conference*, 2004, pp. 31-37.

[5] E. Sanchez, G. Squillero, and A. Tonda, "Post-Silicon Failing-Test Generation through Evolutionary Computation," in *19th IFIP/IEEE International Conference on Very Large Scale Integration*, Hong Kong, 2011.

[6] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: The µGP Toolkit*. Springer, 2010.

[7] C. Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. London: Murray, 1859.

[8] W. Lindsay, E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic test programs generation driven by internal performance counters," in *5th International Workshop on Microprocessor Test and Verification*, 2004, pp. 8-13.

[9] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 369-380, 2001.

[10] B. Colwell, "The Zen of overclocking," *Computer*, vol. 37, no. 3, pp. 9-12, 2004.

[11] A. Chakraborty, et al., "Dynamic Thermal Clock Skew Compensation Using Tunable Delay Buffers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 6, pp. 639-649, 2008.

```
rol ebx, 5
cmp v3, ebx
subpd xmm6, xmm7
mulpd xmm1, [edi+2*16]
or ebx, eax
shl eax, 4
cmp ebx, eax
test v9, ebx
cmp v7, ebx
mov ebx, eax
maxpd xmm3, [edi+3*16]
shl eax, 3
mov eax, v13
sar eax, 7
add ebx, v15
test eax, v16
mov v4, eax
sar eax, 3
pshufb xmm5, xmm2
xor eax, v6
rol ebx, 3
add eax, v16
andpd xmm7, xmm5
xor v13, eax
cmp ebx, v8
phaddd xmm1, xmm1
add ebx, v4
or ebx, v14
shl eax, 9
divpd xmm4, [edi+16]
sar ebx, 4
test eax, v15
mov ebx, v14
andnpd xmm6, xmm5
cmp eax, eax
mov ebx, ebx
shr ebx, 8
sal ebx, 4
rol eax, 4
andpd xmm5, xmm1
shl ebx, 9
and eax, v12
psignd xmm3, xmm7
add ebx, v16
pmulhrsw xmm2, xmm7
sar eax, 4
add v16, eax
sar ebx, 5
mov v11, eax
test v12, eax
sub eax, ebx
pshufb xmm4, [edi+5*16]
andnpd xmm2, xmm5
test v5, ebx
test v4, eax
phsubsw xmm1, [edi]
cmp eax, v11
divpd xmm1, [edi+3*16]
divpd xmm3, xmm4
shr ebx, 0
pshufb xmm3, [edi]
pshufb xmm6, xmm2
haddpd xmm2, xmm4
rol eax, 0
```

Figure 2: Fragment of generated code